NASA Workflow Tool

Workflow Developer Guide

September 29, 2010

1. Overview

The workflow tool automates the steps a user would normally need to do manually. Even without a workflow tool, automation can be a challenge. Throwing in the ability to configure the automation can add to the complications. Designing workflows is not simple.

However, there are a few features and short-cuts provided with the NASA Workflow Tool environment that allow workflow developers to get something working quickly. The first of these is the “Design Mode” feature of the **NASA Experiment Designer** (NED) client software. This allows the developer to create a user-friendly configuration. Next, is the **Light-Weight Workflow Engine** (LWWE) editor software. This allows the developer to specify the various tasks that make up a workflow. Finally, there is a template workflow and examples to help developers get started quickly.

Workflow Parts

For a workflow to run under NED there are several elements that must be defined. At a minimum a developer must create a:

1. **Workflow configuration file** – Contains various options that a developer wants to make available to users (e.g. climate model options and how to display them to the user), as well as metadata about the workflow itself (e.g. the workflow's name and how to retrieve it). This is typically what is displayed to the user when they are setting up a workflow to run. This file supports several formats: compressed ZIP, XML and INI-style.

2. **Workflow architecture file -** This contains all the task information necessary to run a workflow (e.g. building a code and running it). This is typically displayed to the user during the execution of the workflow (after submission). This file also supports several formats: compressed ZIP, XML and INI-style.

Additionally, there are usually other items that must be included in a workflow order to perform more useful work, primarily, system calls. These calls are essentially calls to the system shell to invoke shell commands, executables, scripts, etc.

Light-Weight Workflow Engine

Light-Weight Workflow Engine controls how the workflows run. By specifying tasks and dependencies on those tasks will control the order that things happen. For example, workflow task "B" requires that workflow task "A" run successfully first. So, task "B" depends on the first task before it can start running.

Each task has a type associated with it. The most common type is the SYSTEM_TASK type, which means that the task will invoke a system shell command as you would from the command line.

Additionally, workflow tasks can be grouped into sub-tasks, or child tasks.

- Tasks have a type and can also contain system executable statements.
- Tasks can define dependencies on other tasks.

Running a Workflow

The following are the steps that NED takes in order to run a workflow (see figure below):

1. User submits workflow
    - NED client sends workflow information to NED server
    - Validation is performed by the server
2. A unique working directory is created
3. Configuration files are created
    - This define the experiment variables
    - These are generated according to the NED configuration settings

- The variable files are added to the working directory

4. Workflow scripts are installed
    - Pulled from a source repository
    - Added to the working directory
5. NED server starts the workflow
    - Tasks execute, potentially calling scripts
6. Monitoring occurs
    - Users monitors and controls their workflow

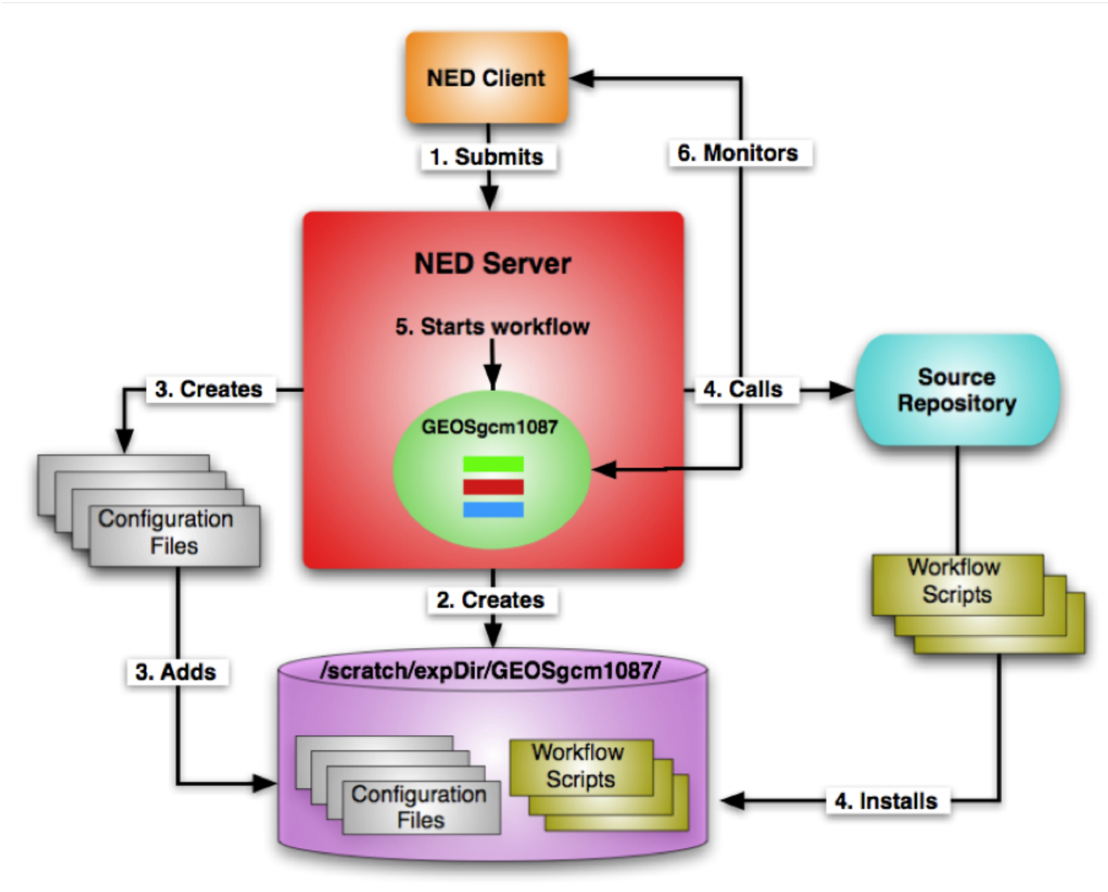

Figure 1. - Steps involved in NED running a workflow

2. Developing a Workflow

Tools Overview

There are two workflow developer tools provided in NED.

- “**Design Mode**” feature of the NASA Experiment Designer (NED) client software. This allows the developer to create a user-friendly configuration.
    - Available through “File / Design Mode” option in NED
    - Makes additional options available:

    - Edit overall workflow properties
    - Add, remove and edit workflow variables and groups
    - Develop configuration scripts for advanced configuration options
    - Example: check for incorrect settings
- **Light-Weight Workflow Engine** (LWWE) Editor. This allows the developer to specify the various tasks that make up a workflow.
    - Allows a workflow developer to quickly construct and validate a workflow architecture
    - Test mode allows the developer to test the workflow without running the actual tasks
    - Shares a common user interface with NED tool
- **Workflow templates and examples**. This allows users to start from a working workflow and expand it into something useful.

Examples and Tutorials

Examples and templates can be obtained from the workflow repository:

```
progress.nccs.nasa.gov/svn/workflows
```

Please refer to the CIB Workflow Workshop 2010 Tutorial on creating a workflow for a quick demo.

LWWE Editor Tool

I. Introduction

The LWWE Workflow Editor is an application for creating, editing, and testing workflows without the need of a workflow server. It uses an embedded workflow engine to execute workflows and has multiple features for easily building and testing a workflow.

Unlike the NED configuration tool, this editor does not provide means for creating or manipulating a NED configuration. This tool is used explicitly for editing, constructing, and testing workflow architectures. Therefore, and configuration files that are necessary to run a workflow must be provided by the workflow developer.

II. User Interface

Most of the user interface should be familiar to those who have used the NED configuration tool.

Major GUI Panels

There are three main panels to the GUI. The first panel displays the “tree” of the currently opened workflow. Depending on the object selected, the right click menu displays different options.

· Rename: Renames the currently selected object.
· Add Task: Only available if the workflow root or task has been selected. This adds a task to the workflow. In the case of a task being selected, the task will be added as a child of that task.

· Remove Task: Only available if a task has been selected. Removes the task from the workflow.
· Add Variable: Only available if the workflow root is selected. Adds a workflow “variable” to the workflow. See the LWWE engine user’s guide for more information in regards to these variables.
· Expand Tree: Expands the workflow tree.

The second major panel is the configuration panel. It serves a dual purpose. First, it provides editing for selected tree objects. Whenever a tree selection is made the configuration panel displays a table containing the properties for that object. From here, a workflow designer can alter the properties as necessary.

The second purpose is to display workflow views of submitted workflows. Whenever a workflow is submitted, a “tree” representation is displayed showing the current status of the workflow and its tasks. This panel also allows control options for the workflow that will be covered later.

The last major panel is the status panel. This displays various status messages from actions taken through design and submission. It also will display log file info for tasks from workflows that have been submitted.

Menu Options
Most menu options are self-explanatory and have short-cut keys to activate them.

File Menu Options
The file menu options contain options for creating, loading, and saving workflow architecture files. This is similar to most other applications.

· New: This will create a very basic default workflow from which to start with.
· Open: Provides a selection dialog to allow the user to select a file to open.
· Save: Saves the current workflow if one is loaded.
· Save As: Provides a selection dialog to allow the user to select or specify a file to save the current workflow to.
· Open Recent: Provides a drop-down list of previously opened workflows. Upon selection, the editor will load the selected workflow (assuming it still exists)
· Exit: Closes the application.

The editor will automatically open the last file the user opened before exiting the application.

Submit Menu Options
The submit menu options provide options for submitting the workflow to run.

· Submit: Submits a workflow for the LWWE engine to execute. This submission mode will execute all logic and executable objects specified by the tasks within the workflow.
· Submit Test: Submits a workflow for the LWWE engine to execute in “test” mode. In test mode, all executable tasks run a “dummy” task. This allows for workflow designers to test the dependency logic of the workflow independent of specific task execution.

After submission, the GUI automatically brings up a workflow status view, which shows the workflow tree and the status of each task. Using the right-click menu can control workflows.

View Menu Options
There is only one view menu option.

· View Workflows: Allows the user to select from a list of previously submitted workflows to view.

Using the right-click menu can control selected workflows.

Help Menu Options
Currently there are no meaningful help options.

III. Workflow View and Control Options

Whenever a workflow is submitted or opened from the View Workflows option, a color-coded tree view is displayed. From this display, right-click menu options can be used to control the workflow.

These options should be familiar to NED users. These options allow workflow level as well as task level controls (starting, stopping, etc.). They also allow for the viewing of log files that are generated by tasks.

3. Advanced Topics

NOTE: Workflow developers are encouraged to share useful tidbits of code. There is a Python workflow library called "marco" that may have utilities and other items useful for workflows and general system functions that aren't necessarily specific to workflows.

Tips: Testing Workflows

- Write your workflow with testing in mind:
- Be generous with logging
- Check for error codes
- Use Python's exception and error handling
- Run the workflow to test it
- New workflows get installed and run under:

/scratch/expWorkDir

- Check the install directory when problems occur
- Use the log files which are viewable from NED
- Test the workflow
- As you development make sure to test frequently
- Use multiple user accounts to avoid hard-wired solutions

Tips: Troubleshooting

When a workflow fails…

- Determine where the problem is coming from
- Files may be missing - did you commit?
- User script errors
- Check the log file
- Edit and re-run the task to see if it passes
- Remote script errors - must have a log file
- Visit Modeling Guru for questions and answers
- NASA website:

http://modelingguru.nasa.gov/clearspace/community/mapmewkflow

- SIVO staff regularly monitors Modeling Guru
- Knowledge base for Earth System modeling
- Contact SIVO workflow staff for more assistance

4. LWWE Architecture Format Reference

I. Introduction

The Lightweight Workflow Engine (LWWE) is a flexible, portable, multipurpose tool for running workflows. Written entirely in Java, the engine can be used either as a stand-alone server or as an embedded component in another application.

Workflows are described in workflow architecture files, which can either be XML or a sectioned text file known as a WIF file. These files describe the task layout and dependencies within workflows. Tasks can use system calls, Java, Jython, or Groovy.

II. The Workflow Architecture File Structure

Every workflow must at least contain workflow architecture file (typically workflowArchitecture.xml or workflowArchitecture.wif). This file describes to the engine the variables, tasks, and task dependencies. The structure of the XML file is based on a JAXB object (Java Binding for XML), which allows for easy loading and saving of the workflow. The file structure of the WIF file is a sectioned file, similar to windows INI files.

The workflow architecture file consists of three parts.

Header
The header contains various attributes about the workflow, such as who created, who submitted it, etc. This is usually located at the top of the file.

Variables
Variables represent quantities that can be referenced within the workflow, added to the environment of system tasks, and other applications.

Tasks
The tasks section contains the information such as task names, dependencies, and other related data. This also can be used to give “structure” to the workflow hierarchy.

The Workflow File Header

The structure of the file was designed for simplicity. The header of the file contains information about the workflow, such as who created it and modified it. Below is a sample header for a workflow architecture file.

XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<WorkflowArchitecture xmlns="LightweightWorkflowEngine">
        <Name>Test Workflow</Name>
        <Description>A test workflow</Description>
        <Creator>Jane</Creator>
        <CreationDate>1/1/1111</CreationDate>
        <ModifiedBy>Joe</ModifiedBy>
        <ModificationDate>1/1/2222</ModificationDate>
   <SubmittedBy>Jim</SubmittedBy>
        <Version>1.0.0</Version>
        <OverallStatus>Completed</OverallStatus>
        <MD5Checksum>abdd00f34500cb</MD5Checksum>
<LoggingDirectory>/log</ LoggingDirectory >
…
```

WIF:

```
[Workflow Architecture]
NAME = Test Workflow
DESCRIPTION = A test workflow
CREATOR = Jane
CREATION_DATE = 1/1/1111
MODIFIED_BY = Joe
MODIFICATION_DATE = 1/1/2222
SUBMITTED_BY = Jim
VERSION = 1.0.0
OVERALL_STATUS = Completed
MD_5_CHECKSUM = abdd00f34500cb
LOGGING_DIRECTORY = /log
```

The first line is just the XML specification. The second line declares the object (WorkflowArchitecture) and the namespace for this file (always LightweightWorkflowEngine). These two lines are the same in every workflow file. The rest of header is described in the following list:

- Name: This is the name of the workflow. The name can be any string.
- Description (optional): A brief description of the workflow.
- Creator (optional): The original creator of the workflow.
- CreationDate (optional): The original creation date of the workflow.
- ModifiedBy (optional): Who last modified this workflow.
- ModificationDate (optional): When the last modification took place.
- SubmittedBy (optional): Who submitted this workflow to execute.
- Version (optional): The version of this workflow.
- OverallStatus: The overall status of this workflow. The engine sets this field.
- MD5Checksum (optional): A checksum of the file.
- LoggingDirectory(optional): The directory in which to write workflow logs for this workflow.

Most for the fields in the header are optional, meaning the workflow will run regardless of what is entered. However, it is a good idea to store meaningful information in these fields as it makes it easier to search and maintain the workflows.

The Workflow File Variables

Variables are simple name-value pairs that can be read and sometimes set within a workflow. The variables are useful when dealing with values that might change during workflow execution, or when tasks need specific environment variables to run.

XML:

```
<WorkflowVariables>
	<Name>X</Name>
	<Value>8</Value>
</WorkflowVariables>
```

WIF:

```
[Variable.X]
NAME = X
VALUE = 8
```

In WIF files, a variable section always takes the form of [Variable.VariableName].

The Workflow File Tasks

The rest of the file is comprised of one or more nested ChildTasks. The following is an example of a ChildTask object.

XML:

```
<ChildTasks>
	<Name>System Task A</Name>
	<Description>A system task</Description>
```

```
        <Information>Some info</Information>
        <RetriesOnFail>0</RetriesOnFail>
        <TaskStatus>Completed</TaskStatus>
        <TaskType>System</TaskType>
        <IterationLimit>0</IterationLimit>
        <Iteration>0</Iteration>
        <ExecutableObject>ls -lrt</ExecutableObject>
        <TaskDependency>Task A eq Completed</TaskDependency>
        <CompletionDependency>Task C eq Completed</CompletionDependency>
        <TimeDependency>TIME(2/2/2009:12:15:00)</TaskDependency>
</ChildTasks>
```

WIF:

```
[Task.Path.To.Task]
NAME = Task
DESCRIPTION = The task
INFORMATION =
TASK_TYPE = SYSTEM
TASK_STATUS = QUEUED
RETRIES_ON_FAIL =
ITERATION_LIMIT = 1
ITERATION = 1
EXECUTABLE_OBJECT =
TASK_DEPENDENCY =
TIME_DEPENDENCY =
COMPLETION_DEPENDENCY =
```

The properties of a ChildTasks object are described below:

- Name: Every task must have a UNIQUE name that identifies it. Pretty much any string can be used for a name, including names with spaces.
- Description (optional): A brief description of this task.
- Infromation (optional): Any additional information about the task. Unlike Description, tasks and/or the engine dynamically update this field.
- RetriesOnFail (optional): The number of times to retry an aborted task automatically. This is currently ignored.
- TaskStatus: The status of the task. Initially this can be absent or empty, however the workflow engine automatically populates this field when running.
- TaskType: The type of this task. This will be covered more in depth later.
- IterationLimit (optional): The maximum number of times this task can be executed. A missing value or value of <= 0 indicates that there is no limit. See the Task Looping section for more information.
- Iteration (optional): The current number of times this task has been executed. See the Task Looping section for more information.
- ExecutableObject: The actual object executed by this task when the task and time dependencies are satisfied. Depending on the task type this can be anything from a command line specification to a Java class name to nothing at all.

- TaskDependency (optional): A logical “equation” to evaluate that contains task dependencies for this task. This is covered in depth in the Task Dependency section.
- CompletionDependency (optional): A logical “equation” to evaluate that contains dependencies for determining whether the task is complete. Some task types utilize this field.
- TimeDependency (optional): A time dependency specification for this task. This is covered later in depth in the Time Dependency section.

Note that for the WIF file format, nesting is determined by the section name (what appears between the brackets. All task section names must begin with Task. So to declare Task B under Task A, the section name would be [Task.Task A.Task B].

Task Type

The first important field is TaskType. There are several different task types within the engine. Some task types are built-in while others allow for extensions or customized tasks.

The following is a list of the currently supported built-in types and the summary of their purpose.

· **System**: The system type task executes an operating system command. This includes executing native scripts.
· **Parent**: This task acts as a way to group related children tasks together. The parent task acts a monitor for the child tasks, reflecting the worst state of the children.
· **Loop**: This task is similar to the parent task, with the addition that it will loop over the child tasks for a fixed number of times.
· **Timespan**: This task type is like the loop task, except it uses date-time iterators instead of a simple index count.

The other task types (**Java**, **Jython**, and **Groovy**) allow for programmatic customization of a task. These will be covered in the API documentation.

Task Modifiers

Along with task types, there are also task modifiers. A task modifier influences the behavior of a task or how it interacts with the overall workflow.

The following are the current task modifiers:

MANUAL_TASK: If this flag is specified in the task dependencies, this task will be treated as a manual task. A manual task is a task that can ONLY be triggered by the user. Other tasks can depend on this task, and this task’s status will be reflected when determining the overall workflow status.

OPTIONAL_TASK: If this flag is specified in the task dependencies, this task will be treated as an optional manual task. An optional manual task can only be triggered by user action. Unlike the manual task, this task has no effect on the overall workflow status unless it is invoked. Also, having other tasks depend on this task will always result in a true status, effectively rendering the dependency useless.

Executable Object

The ExecutableObject contains the specification for what is to be executed by the task. This varies by task type, and in some cases is ignored.

Among the built-in task types, only system tasks utilize this field. An OS command is specified here and will be executed when the task runs. For other built-in task types, this field is not used.

For the scripting task types this can either represent a class or script file. This will be covered more in depth in the API documentation.

Task Dependency

The TaskDependency is an optional field that is the logic equation used to determine when a task may be executed during a workflow. The logic equation is used to evaluate whether or not all task dependencies have been met for this task. If there is no string specified, then the task will run when the workflow is started (assuming it doesn't have a time dependency).

The grammar for the task dependency string is simple. Typically, you want to check on task status conditions for various tasks and execute a task when those conditions are met. For example, you may want Task B to run only if Task A has been completed successfully, as shown in the example above. The equation for this is simply:

Task A eq Completed

Note that all task names are case sensitive. Logical operators and status conditions are not case sensitive.

For testing task conditions, there are two acceptable operators:

- eq: Checks to see if the status of the specified task is equal to the specified status.
- neq: Checks to see if the status of the specified task is not equal to the status specified.

The engine also supports logical concatenation operators for performing more than one dependency check:

- and: Performs a logical and between two conditions
- or: Performs a logical or between two conditions
- xor: Performs a logical exclusive or between two conditions

The logical concatenators are for evaluating multiple dependencies. For example, a task with two dependencies may look like:

Task B eq Completed and Task A eq Completed

You can also use parentheses to set the order of operations:

Task C eq Completed or (Task A eq Completed and Task B eq Completed)

To the engine, this reads as “If Task C is completed or if Task A and Task B are completed”.

The task status conditions can be any of the following:

- Completed: The task completed successfully
- Aborted: The task was aborted
- Suspended: The task was suspended
- Resumed: The task was resumed
- Updated: The task was updated
- Queued: The task was queued
- Running: The task was running

So another task dependency may look like this:

Task B eq Completed or Task A eq Aborted

Which can be read as "Run this task if Task B is completed or if Task A failed".

NOTE: The built-in container type tasks (Parent, Loop, Timespan) ignore task dependencies. Manual and optional tasks also ignore this field.

Completion Dependency

A completion dependency is an optional field that can specify when a particular task should be marked as complete. The rules for a completion dependency are the same as for a task dependency.

The only built-in type that currently supports completion dependencies are system tasks.

NOTE: Container type tasks (Parent, Loop, Timespan) inherently depend on their children for completion status. Any completion dependencies are ignored. Manual and optional tasks also ignore this field.

Time Dependency

In addition to a task dependency, there can also be a time dependency. Time dependencies are instigated after task dependencies have been satisfied. The format here is also simple. Here is an example:

TIME(2/2/2009:12:15:00)

All time dependencies have the format TIME(...). A time dependency can have up to three arguments. The following are the acceptable time dependency specifications:

- TIME (datetime): Specifies a time dependency that will fire at the specified date and time.
- TIME (datetime, interval): Specifies a time dependency that begins at a specified date and re-triggers after every interval.
- TIME (datetime, interval, delay): Specifies a time dependency that begins at a specified date. After the delay, it will re-trigger at the specified interval.

Using * for the datetime will automatically use the current system time. So a time dependency specified like this:

TIME(*, 3600)

Would set up a time dependency that would repetitively trigger after every hour, using the current time as the starting point.

NOTE: The built-in container type tasks (Parent, Loop, Timespan) ignore time dependencies. Manual and optional tasks also ignore this field.

III. Built-In Task Types

As noted, there are several different built-in task types that can be used to quickly construct a workflow.

The System Task

The simplest task type is System, as shown in the example.

XML:

```
<ChildTasks>
        <Name>System Task A</Name>
   <Description>A system task</Description>
        <Information>Some info</Information>
        <TaskType>System</TaskType>
        <ExecutableObject>ls -lrt</ExecutableObject>
        <TaskDependency>Task A eq Completed</TaskDependency>
</ChildTasks>
```

WIF:

```
[Task.System Task A]
NAME = System Task A
DESCRIPTION = A system task
INFORMATION = Some info
TASK_TYPE = System
EXECUTABLE_OBJECT = ls -lrt
TASK_DEPENDENCY = Task A eq Completed
```

A System task is a fully automated task that runs whatever command is entered in the ExecutableObject field. In the example, this is simply a command line call to ls, though it could be anything that can be run on the command line, including bash and python scripts. The engine handles all task status setting and updating. For an initial attempt at a workflow, the System type is the easiest to use.
All system type tasks automatically create a log file which stores all stderr and stdout output from the executing process.

Other task types are free to implement their own form of logging, or none at all if it is deemed unnecessary.

As noted, a System type task is a special task that runs a command on the command line. The call to the system is independent of the workflow engine. Other than starting the process and analyzing the return code, there is no direct interaction with the running task.

However, occasionally there may be need to have more information than just the state of a task to be sent to a user or stored with the workflow. In these instances, a workflow designer can have a task output messages with special keywords to indicate to the engine that the user should be notified or the information should be stored.

The keywords **MUST** appear at the start of the line and be terminated with a colon character. The keywords are case sensitive. The following is a list of keywords, their meanings, and how the engine handles the message.

- FATAL: Indicates a fatal problem. The string is used to create a status message, which is sent back to anyone monitoring the workflow.
- ERROR: Indicates an error. The string is used to create a status message, which is sent back to anyone monitoring the workflow.
- WARN: Indicates a warning. The string is used to create a status message, which is sent back to anyone monitoring the workflow.
- INFO: Indicates an information message. The string is used to create a status message, which is sent back to anyone monitoring the workflow.
- DEBUG: A debugging message. The string is used to create a status message, which is sent back to anyone monitoring the workflow. These should be removed before a release.
- UPDATE: Indicates that a task's information should be updated. Unlike the other keywords, this does not create a status message. Instead, the information is used to update the task information. This information is stored with the workflow.

Some examples of parse-able messages are:

UPDATE: I'm running my second loop!
ERROR: No directory was found!
WARN: Value X was 3 but should have been 4.

The Loop Task

The loop task provides a convenient way to have a loop within a workflow.

XML:

```
<ChildTasks>
	<Name>Loop Task</Name>
	<Description>A looping task</Description>
	<Information>Some info</Information>
	<IterationLimit>10</IterationLimit>
	<TaskType>Loop</TaskType>
</ChildTasks>
```

WIF:

```
[Task.Loop Task]
NAME = Loop Task
DESCRIPTION = A looping task
INFORMATION = Some info
ITERATION_LIMIT = 10
TASK_TYPE = Loop
```

Unlike the System task, the loop task does not use an ExecutableObject (if one is specified, it is ignored) or dependency fields. The task starts executing when the workflow starts and assumes the worst-case state of its children. The task runs until the iteration limit is reached, reflecting the worst-case status of the children it is looping over. For example, even though the loop task is running it will not be marked as being in the running state until the worst-case status of it's children is “running”.

NOTE: Do not use the Loop Task state as a dependency in child tasks. The Loop Task state does not reflect the state of the Loop Task; it reflects the state of the child tasks. Therefore, setting up a dependency on the Loop Task itself may not yield the expected behavior.

The only “safe” states that can be used for dependencies for tasks outside the loop are the completion states (COMPLETED and ABORTED). These states indicate that the loop is no longer running. While the loop is running, the state will be unstable and should not be relied upon.

The Loop task triggers a new iteration when all it's children have reached a completed state. The loop task will abort itself if any child becomes aborted. An example of a loop using the looping task follows.

XML:

```
<ChildTasks>
	<Name>Loop Task</Name>
	<Description>A looping task</Description>
	<Information>Some info</Information>
	<IterationLimit>10</IterationLimit>
```

```
        <TaskType>Loop</TaskType>
        <ChildTasks>
           <Name>Some Task</Name>
           <Description>Some task</Description>
           <TaskType>System</TaskType>
         <ExecutableObject>ls -lrt</Executable>
           <TaskDependency></TaskDependency>
        </ChildTasks>
</ChildTasks>
```

WIF:

```
[Task.Loop Task]
NAME = Loop Task
DESCRIPTION = A looping task
INFORMATION = Some info
ITERATION_LIMIT = 10
TASK_TYPE = Loop

[Task.Loop Task.System Task A]
NAME = System Task A
DESCRIPTION = A system task
TASK_TYPE = System
EXECUTABLE_OBJECT = ls -lrt
```

In the above example, the loop task will iterate whenever Some Task completes. The loop task finally completes (marks itself as completed) when this has happened 10 times.

While specifying a limit can be useful in some cases, often a loop limit may need to be determined “dynamically”. The iteration limit field can take a workflow variable name as well. For example, if there is a variable known to the workflow engine called X and it is numeric, then you can specify the iteration limit as:

XML:

```
<IterationLimit>X</IterationLimit>
```

WIF:

```
ITERATION_LIMIT = X
```

In this case, when this task is executed it will look up the variable value and use that for the iteration limit. The variables that a workflow can access are implementation specific. Whatever is using the workflow engine is responsible for setting up accessible variables. It is also possible to specify variables directly within the workflow architecture file.

Parent Task

The parent task is a specialized task that is designed to act as a parent for a group of related tasks.

The parent task, like the loop task, does not use an ExecutableObject. Instead, it relies on its children to determine when the task is complete. Like the loop task, the parent task starts as soon as the workflow starts.

The parent task marks itself as complete only when all of its children have been completed. While the task is waiting, it reflects the worst-case status of its children.

The parent task ignores any specified dependencies.

XML:

```
<ChildTasks>
        <Name>Parent Task</Name>
        <Description>A container task</Description>
        <Information>Some info</Information>
        <TaskType>Parent</TaskType>
        <ChildTasks>
          <Name>Task A</Name>
          <Description>Some task</Description>
          <TaskType>System</TaskType>
         <ExecutableObject>ls -lrt</Executable>
          <TaskDependency>Some Task eq Running</TaskDependency>
        </ChildTasks>
</ChildTasks>
```

WIF:

```
[Task.Parent Task]
NAME = Parent Task
DESCRIPTION = A looping task
INFORMATION = Some info
TASK_TYPE = Parent

[Task.Parent Task.Task A]
NAME = Task A
DESCRIPTION = A system task
TASK_TYPE = System
EXECUTABLE_OBJECT = ls -lrt
TASK_DEPENDENCY = Some Task eq Running
```

Task A starts running when Some Task starts running. The container task will remain in the running state until Task A completes.

NOTE: Do not use the Parent Task state as a dependency in child tasks. The Parent Task state does not reflect the state of the Parent Task; it reflects the state of the child tasks.

Therefore, setting up a dependency on the Parent Task itself may not yield the expected behavior.

Like the loop task, the only “safe” states that can be used for dependencies for tasks outside the loop are the completion states (COMPLETED and ABORTED). These states indicate that the parent is no longer running. While the parent is running, the state will be unstable and should not be relied upon.

Java, Groovy, and Jython Tasks

These task types allow a designer to specify a compiled class or script that has been derived from the workflow task API classes as tasks within a workflow.

This offers the utmost in flexibility, as new functionality can be dynamically added to the workflow engine and its tasking capability. However, these tasks require programming knowledge and familiarity with the workflow engine API, so these will be covered in the API documentation.

IV. Other Aspects

Nested Tasks

There are no limits to the depth of nesting for tasks. However, in practice having heavily nested tasks can lead to an unwieldy file if you’re using the XML format. The WIF format does not experience this problem, but task sections names become long.

While XML lends itself to a hierarchical structure, that structure is superficial to the workflow itself. Aside from the container type tasks like parent and loop tasks, task execution is determined by dependencies, not their placement within the workflow file or how deeply they are nested. However, it is a good practice to place tasks where they logically would occur during workflow execution.

The same applies for WIF types files as well. While the dependencies will always ensure correct execution order, it is easier to manage the file if tasks appear in a logical order.

Another thing to keep in mind is that while technically any task can have nested tasks, it is only really meaningful if the tasks are children of a “container” type task (Parent, Loop, Timespan).

So while it is possible to have system type tasks nested within system type tasks, it is not a good way to represent a workflow and carries no significance in how the workflow will be executed.

Task Looping Without Using Loop

It is possible to construct a loop in a workflow without using the Loop task, but it is not recommended due to how easily bugs can be introduced and how hard they can be track down.

XML:

```
<ChildTasks>
        <Name>Task A</Name>
        <Description>A task</Description>
        <TaskType>System</TaskType>
        <ExecutableObject>ls -lrt</ExecutableObject>
        <TaskDependency></TaskDependency>
</ChildTasks>
<ChildTasks>
        <Name>Task B</Name>
        <Description>A looping task</Description>
        <TaskType>System</TaskType>
        <ExecutableObject>ls -lrt</ExecutableObject>
        <TaskDependency>Task A eq Completed or Task C eq Completed</TaskDependency>
        <ChildTasks>
                <Name>Task C</Name>
                <Description>Another task</Description>
                <TaskType>System</TaskType>
                <ExecutableObject>ls -lrt</ExecutableObject>
                <TaskDependency>Task B eq Completed</TaskDependency>
        </ChildTasks>
</ChildTasks>
```

WIF:

```
[Task.Task A]
NAME = Task A
DESCRIPTION = A task
TASK_TYPE = System
EXECUTABLE_OBJECT = ls -lrt
TASK_DEPENDENCY = Some Task eq Running

[Task.Task B]
NAME = Task B
DESCRIPTION = A looping task
TASK_TYPE = System
EXECUTABLE_OBJECT = ls -lrt
TASK_DEPENDENCY = Task A eq Completed or Task C eq Completed

[Task.Task B.Task C]
NAME = Task C
DESCRIPTION = Another task
TASK_TYPE = System
EXECUTABLE_OBJECT = ls -lrt
TASK_DEPENDENCY = Task B eq Completed
```

As can be seen, Task A has no dependencies, so starts when the workflow is executed. Task B will trigger whenever Task A is completed or whenever Task C is completed. Task C is triggered whenever Task B is completed.

When the workflow runs, Task A will execute to completion. Barring any errors, it will notify Task B that it has completed. This will trigger Task B to run. When Task B completes, it will notify Task C. When Task C completes, it will again notify Task B to run.

An even simpler loop can be made:

XML:
```
<ChildTasks>
        <Name>Task A</Name>
        <Description>A task</Description>
   <TaskType>System</TaskType>
        <ExecutableObject>ls -lrt</ExecutableObject>
        <TaskDependency>Task A eq Completed</TaskDependency>
</ChildTasks>
```

WIF:
```
[Task.Task A]
NAME = Task A
DESCRIPTION = A task
TASK_TYPE = System
EXECUTABLE_OBJECT = ls -lrt
TASK_DEPENDENCY = Task A eq Completed
```

Task A depends only on itself completing, and will continuously retrigger every time it completes.

The looping described above is infinite. There are no limits on how many times the tasks will be called. For better control, a workflow designer would use the IterationLimit field.

By specifying the iteration limit, the engine will only execute a task up to the iteration limit. After that, the task remains in its completed state and no longer sends out triggering information unless forced to by a user action.

The current iteration of a task is tracked ONLY if the IterationLimit has been set. The field used for tracking the current iteration is the Iteration field.

So within a workflow file, an iterative task may look like this:

XML:
```
<ChildTasks>
        <Name>Task A</Name>
        <Description>A task</Description>
        <TaskType>System</TaskType>
```

```
        <IterationLimit>10</IterationLimit>
    <Iteration>3</Iteration>
        <ExecutableObject>ls -lrt</ExecutableObject>
        <TaskDependency>Task A eq Completed</TaskDependency>
</ChildTasks>
```

WIF:

```
[Task.Task A]
NAME = Task A
DESCRIPTION = A task
TASK_TYPE = System
EXECUTABLE_OBJECT = ls -lrt
ITERATION_LIMIT = 10
ITERATION = 3
TASK_DEPENDENCY = Task A eq Completed
```

This field is used by the engine, and is stored with the rest of the workflow. While manipulation of this field is possible, it is not recommended.

It is strongly recommended to use the loop task instead of manipulating task dependencies to create loops.

V. Workflow Execution

When a workflow architecture file is loaded by the engine, it first goes through and loads all the tasks. Then the engine goes through each task, determines the dependencies, sets up the necessary notifications, and waits to receive a run command.

When a run command is received, the engine broadcasts a start message to the workflow. Any tasks that have no dependencies are immediately triggered to run. **In every workflow there must be at least one task that has no dependencies, otherwise the workflow will not automatically start**. Otherwise, the workflow will wait for a user invocation to start.

An engine user may also set “environment” values into the engine. These values are accessible directly via the API for Java, Groovy, and Jython tasks. For System tasks, these variables are added as process environment variables. This functionality is useful for setting common variable values that are used throughout the workflow, such as a user name or a group code. The System type tasks however are not able to alter these variables other than in its process.

IV. LWWE Appendix

XML Workflow Example:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<WorkflowArchitecture xmlns="LightweightWorkflowEngine">
```

```
<Name>LWWE_null</Name>
<Description>A workflow for testing the workflow engine</Description>
<Creator>ME</Creator>
<CreationDate>1/1/1111</CreationDate>
<ModifiedBy>YOU</ModifiedBy>
<ModificationDate>1/1/2222</ModificationDate>
<SubmittedBy>someone</SubmittedBy>
<Version>1.0.0</Version>
<OverallStatus></OverallStatus>
<MD5Checksum>0</MD5Checksum>
<ChildTasks>
<Name>Task A</Name>
<Description>A system task</Description>
<TaskType>System</TaskType>
<ExecutableObject>ls -lrt</ExecutableObject>
</ChildTasks>
<ChildTasks>
<Name>Parent Task B</Name>
<Description>A parent task</Description>
<TaskType>Parent</TaskType>
<!—This is a child task of parent task b à
<ChildTasks>
<Name>Task C</Name>
<Description>A system task</Description>
<TaskType>System</TaskType>
<IterationLimit>10</IterationLimit>
<ExecutableObject>echo “Task C says hello repeatedly!”</ExecutableObject>
<TaskDependency>Task A eq Completed</TaskDependency>
</ChildTasks>
</ChildTasks>
</WorkflowArchitecture>
```


WIF Workflow Example:

```
[Workflow Architecture]
NAME = LWWE__2010_Jun_23_15_36_49_0973
VERSION = 1.0.0
CREATOR = ME
CREATION_DATE = 1/1/1111
MODIFIED_BY = ME
MODIFICATION_DATE = Wed Jun 23 15:36:44 EDT 2010
SUBMITTED_BY = ME
DESCRIPTION = A workflow for testing the workflow engine
MD_5_CHECKSUM = 0
LOGGING_DIRECTORY =
OVERALL_STATUS =

[Variable.X]
NAME = X
VALUE = 1

[Variable.Y]
NAME = Y
```

```
VALUE = 2

[Variable.Z]
NAME = Z
VALUE = 3

[Task.System Task A]
NAME = System Task A
DESCRIPTION = A system task
INFORMATION =
TASK_TYPE = SYSTEM
TASK_STATUS = COMPLETED
RETRIES_ON_FAIL =
ITERATION_LIMIT =
ITERATION = 0
EXECUTABLE_OBJECT = echo "Hi from Task A"
TASK_DEPENDENCY =
TIME_DEPENDENCY =
COMPLETION_DEPENDENCY =

[Task.Loop Task]
NAME = Loop Task
DESCRIPTION = Loop Task
INFORMATION = Loop Task [Run 3 of 3]
TASK_TYPE = LOOP
TASK_STATUS = COMPLETED
RETRIES_ON_FAIL =
ITERATION_LIMIT = 3
ITERATION = 3
EXECUTABLE_OBJECT =
TASK_DEPENDENCY =
TIME_DEPENDENCY =
COMPLETION_DEPENDENCY =

[Task.Loop Task.Parent Task 1]
NAME = Parent Task 1
DESCRIPTION = Parent task 1
INFORMATION =
TASK_TYPE = PARENT
TASK_STATUS = COMPLETED
RETRIES_ON_FAIL =
ITERATION_LIMIT =
ITERATION = 0
EXECUTABLE_OBJECT =
TASK_DEPENDENCY =
TIME_DEPENDENCY =
```

COMPLETION_DEPENDENCY =

[Task.Loop Task.Parent Task 1.System Task C]
NAME = System Task C
DESCRIPTION = System task
INFORMATION =
TASK_TYPE = SYSTEM
TASK_STATUS = COMPLETED
RETRIES_ON_FAIL =
ITERATION_LIMIT =
ITERATION = 0
EXECUTABLE_OBJECT = echo "Hi from Task C"
TASK_DEPENDENCY = System Task A eq Completed
TIME_DEPENDENCY =
COMPLETION_DEPENDENCY =

[Task.Loop Task.Parent Task 1.System Task D]
NAME = System Task D
DESCRIPTION = System task
INFORMATION =
TASK_TYPE = SYSTEM
TASK_STATUS = COMPLETED
RETRIES_ON_FAIL =
ITERATION_LIMIT =
ITERATION = 0
EXECUTABLE_OBJECT = echo "Hi from Task D"
TASK_DEPENDENCY = System Task C eq Completed
TIME_DEPENDENCY =
COMPLETION_DEPENDENCY =

[Task.Manual Task MT]
NAME = Manual Task MT
DESCRIPTION = A manual task
INFORMATION =
TASK_TYPE = SYSTEM
TASK_STATUS = COMPLETED
RETRIES_ON_FAIL =
ITERATION_LIMIT =
ITERATION = 0
EXECUTABLE_OBJECT = echo "Hi from Manual Task MT"
TASK_DEPENDENCY = MANUAL_TASK
TIME_DEPENDENCY =
COMPLETION_DEPENDENCY =

[Task.Optional Task OT]
NAME = Optional Task OT

```
DESCRIPTION = An optional task
INFORMATION =
TASK_TYPE = SYSTEM
TASK_STATUS = QUEUED
RETRIES_ON_FAIL =
ITERATION_LIMIT =
ITERATION = 0
EXECUTABLE_OBJECT = echo "Hi from Optional Task OT"
TASK_DEPENDENCY = OPTIONAL_TASK
TIME_DEPENDENCY =
COMPLETION_DEPENDENCY =
```